プログラミングの基礎（makeを使った分割コンパイルと並列処理）

2010年3月17日

鴨志田 良和(東京大学情報基盤センター)

目次

- 使用可能なファイルシステム
- バッチキューの操作(上級編)
- makeを使った分割コンパイル
- makeを使った並列処理

この講習の目的

- HA8000クラスタシステムにログインして効率的に作業を行えるようになることを目指し、
  - ファイルシステムやキューの操作について学ぶ
- 大規模なプログラムを作成する際に必須となる、分割コンパイルの方法について学ぶ
- makeを使用した並列処理の方法について学ぶ

ファイルシステム

利用可能なファイルシステム

- HA8000クラスタシステムで利用可能なファイルシステムは以下のとおりである

PATH	種類	共有/非共有
/home/ログイン名	HSFS home1～4に分割	共有
/short/ログイン名	HSFS	共有
/nfs/all/ログイン名	NFS	共有
/lustre/ログイン名	Lustre	共有
/tmp	ローカルディスク	非共有

目的に応じて使い分けると効率的に作業ができる

ファイルシステム選択の基準

- 分散ファイルシステム(HSFS, Lustre)
  - 複数のファイルサーバにデータを分散できるため、多くのプロセスからアクセスする場合に効率がよい
- NFS
  - ログインノードにおける作業時やファイル操作のレスポンスが重視される場合に適している
  - 負荷分散機能がない
- ローカルディスク
  - /tmpに置いたファイルは、ログインノードでは1～2日で、計算ノードではジョブの終了時に削除される
  - 他のノードから直接アクセスできない

分散ファイルシステム

- HSFS(Hitachi Striping Filesystem)
  - 複数ノードから大きなブロックサイズ(数MB以上?)で入出力を行う場合に効率がよいとされている
  - 現在はノードあたりの最大帯域幅は1Gbps
    - 今後改善の計画あり
  - /shortは5日後に削除される(一時的なデータ置き場)
- Lustre
  - 2010年3月からサービス開始
  - 大規模ファイル入出力、メタデータ操作の両方で高性能なファイルシステム
  - データの分散方法をファイルごとに指定可能(後述)

利用可能な容量(quota)

- 共有ファイルシステムは、個人、またはグループに対して利用可能容量の制限(quota)がある
- 残り容量の確認コマンド

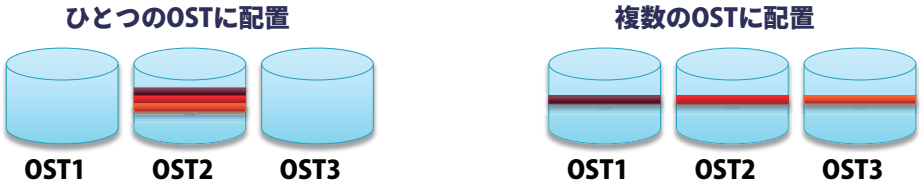
HSFS	`la -a`
NFS	`quota -v`
Lustre	`lfs quota /lustre/`
/tmp	`df /tmp`

課題1

- **それぞれのファイルシステムでファイル展開コマンドを実行せよ**
  - **実行時間にどのような差があるか？**
- **各ファイルシステムの残り容量を確かめよ**

Lustreのデータ配置の指定

- **データ配置の指定**
  - **HA8000のLustreファイルシステムは30個のOST(Object Storage Target: 仮想的なディスク)で構成される**
  - **データをひとつのOSTに配置するか、複数のOSTに分散して配置するかはユーザが指定できる**
  - **デフォルトではひとつのOSTに配置**
  - **lfs getstripe / lfs setstripeコマンドで参照・変更可能**



Lustreのデータ配置の指定(例)

- **lfs setstripe -s *size* -c *count* ファイル名**
  - ***size* 毎に*count* 個のOSTに渡ってデータを分散配置する設定にした空のファイルを作成する**

```
$ dd if=/dev/zero of=/lustre/$USER/4G.dat bs=1M count=4096
4096+0 records in
4096+0 records out
4294967296 bytes (4.3 GB) copied, 41.0291 seconds, 105 MB/s
                                        OST数が1の場合の書き込み性能
$ rm /lustre/$USER/4G.dat
$ lfs setstripe -s 1M -c 4 /lustre/$USER/4G.dat
                          ストライプ設定の変更(4つのOSTにデータを分散)
$ dd if=/dev/zero of=/lustre/$USER/4G.dat bs=1M count=4096
4096+0 records in
4096+0 records out
4294967296 bytes (4.3 GB) copied, 10.9823 seconds, 391 MB/s
                                        OST数が4の場合の書き込み性能
```

キュー操作上級編(1/2)

- **qstat -l ジョブID (qstat -f)**
  - **ジョブの、より詳しい状態を確認するコマンド**
  - **ジョブIDを指定しない場合は実行前・実行中の、自分のすべてのジョブが対象**
- **qscript ジョブID**
  - **投入したスクリプトを確認するコマンド**

qstat -fの出力例

```
BATCH REQUEST:  253443.batch1
    Name:  test.sh
    Owner: uid=33071, gid=30144
    Priority: 63
    State: 1(RUNNING)
    Created at:  Tue Jun 30 05:36:24 2009
    Started at:  Tue Jun 30 05:36:27 2009
    Remain : 14 minutes 47 seconds

QUEUE
    Name:  lecture5

RESOURCES
    Per-proc. CPU time limit      = UNLIMITED
                          warning = 0 seconds
    Per-req.  CPU time limit      = UNLIMITED
                          warning = 0 seconds
    Per-proc. core file size limit= 0 kilobytes
    Per-proc. data size limit     = 28 gigabytes
                          warning = 0 kilobytes
    Per-proc. perm file size limit= UNLIMITED
                          warning = 0 kilobytes
    Per-proc. memory size limit   = 28 gigabytes
    Per-req.  memory size limit   = 28 gigabytes
    Per-proc. stack size limit    = 2 gigabytes
    Per-proc. volafile size limit = 0 kilobytes
    Per-req.  volafile size limit = 0 kilobytes
    Per-proc. working set limit   = 28 gigabytes
    Per-req.  etime limit         = 15 minutes
                          warning = 0 seconds
    Per-proc. execution nice pri. = 0

FILES
    Stderr: None
    Stdout: None
    stderr to stdout: No

MAIL
    Address: t35001@ha8000-3.cc.u-tokyo.ac.jp
    When:

MISC
    Partition: None
    Node: 1
    Jobtype: T1
    Rerunnable: Yes
    Performance information: Yes
    Shell: None
    Account name: f
    Qsub at: /nfs/all/t35001
```

キュー操作上級編(2/2)

- qsub -N 2 -q debug **スクリプト名**
  - **debugキューの2ノードを使用して実行**
  - **ジョブスクリプトに書いたものより、コマンドライン引数で指定したオプションのほうが優先される**
  - **注意:** qscript**で表示されるものと実際のオプションが異なる場合がある**
  - qstat -f**を使って確認すれば、正しい情報が得られる**

課題2

- qsub -q lecture7**コマンドを実行し、標準入力に** env|sort; sleep 30**を入力してCtrl-Dキーで終了**
- qscript**でスクリプトを確認せよ**
- qstat -l**で詳細情報を確認せよ**
- **ジョブ終了後、**STDIN.o??????**に出力された内容を確認せよ**
  - **どのような環境変数が設定されているか**

課題3

- **実行されたジョブのノード数(**-N**)とジョブタイプ(**-J**)を標準出力に表示するスクリプトを書け**

makeで分割コンパイル

make

- プログラムの分割コンパイル等を支援するソフトウェア
- 変更があったファイルのみを再コンパイル
- 大規模なプログラムを書くときに便利
- 本質的にはワークフロー言語の実行エンジン
- コンパイルに限らず、処理の依存関係を記述して、依存関係に従ってコマンドを実行できる
  - この講習会ではGNU make (version 3.81)を使用する

Hello, world!

- **hello.c**

```
#include <stdio.h>
int main(int argc, char** argv) {
  printf("Hello, world!¥n");
  return 0;
}
```

- **Makefile**

```
hello: hello.c
	gcc -o hello hello.c
```

  - スペースでなくタブにする

- **実行**

```
$ make hello
gcc -o hello hello.c
```

もう一度makeを実行するとどうなるか？

```
$ make hello
make: `hello' is up to date.
```

Makefileの構造

- ルールは、ターゲット、依存するファイル、コマンドで記述される

```
ターゲット: 依存するファイル …
	コマンド
	…
```

- **makeの実行**
  - make ターゲット
  - ターゲットを省略した場合は、Makefileの最初のターゲットが指定されたものとして実行される

コマンドが実行される条件

- 以下のいずれかが満たされる場合にコマンドを実行
  - ターゲットが存在しない
  - (ターゲットのタイムスタンプ)
    < (依存するいずれかのファイルのタイプスタンプ)
- 依存するファイルXが存在しない場合、make Xを先に実行
- コマンドを実行した後の終了ステータスが0以外の場合は続きの処理を実行しない

少し複雑な例

- **hello.c**

```
#include <stdio.h>
void hello(void) {
  printf("Hello, world!¥n");
}
```

- **main.c**

```
void hello(void);
int main(int argc, char** argv) {
  hello();
  return 0;
}
```

- **Makefile**

```
hello: hello.o main.o
	gcc -o hello hello.o main.o
hello.o: hello.c
	gcc -c hello.c
main.o: main.c
	gcc -c main.c
```

- **実行**

```
$ make
gcc -c hello.c
gcc -c main.c
gcc -o hello hello.o main.o
```

- **hello.cを書き換え**

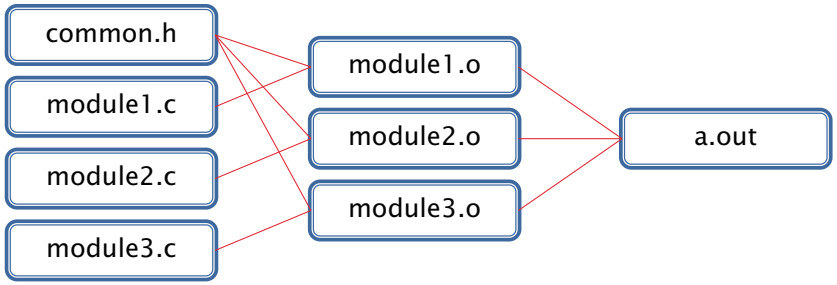
例: world! を world!! に書き換え

- **makeを再実行**

```
$ make
gcc -c hello.c
gcc -o hello hello.o main.o
```

分割コンパイル

- 2回目のmakeで起きていたこと
  - main.oのコンパイルは、main.cに変更がなかったため行われなかった
- Makefileに依存関係を適切に記述することで、変更があった部分だけを再コンパイルすることができる



Makeのtips

- **Makefileの指定**

```
$ make -f test.mk
```

- **長い行**

```
hello: hello.o main.o
	gcc -g -Wall -O3 ¥
	  -o hello hello.o main.o
```

- **PHONYターゲット**

```
.PHONY: clean
clean:                (cleanというファイルがあっても実行する)
	rm -f hello hello.o main.o
```

課題4

- **コマンドの前のタブを、スペースにした場合、どのようなエラーが出力されるか**
- **.PHONY: Xがあるときとない時で、make Xの動作に違いがあることを確認せよ**

高度なMakefileの書き方

- **変数、関数の使用・特別なルールの書き方**

- **Makefileのより簡潔な記述**
- **より柔軟な出力やエラー制御**

変数の使い方

- **代入方法**

```
OBJECTS=main.o hello.o
```

- **参照方法**

```
hello: $(OBJECTS)
```

${OBJECTS}でもよい
$OBJECTSとすると、$(O)BJECTSと同じことになる

- **再帰的な展開**

```
CFLAGS=$(INCLUDES) -O -g
INCLUDES=-Idir1 -Idir2
```

CFLAGSは -Idir1 -Idir2 -O -gに展開される

makeの動作の制御

- **実行しようとするコマンドを表示しない**

```
test1:
	@echo Test message
```

- **コマンド終了時ステータスを無視する**

```
test2:
	-rm file1 file2 file3
```

条件分岐

- **コマンドの条件分岐**

```
hello: $(OBJECTS)
ifeq ($(CC),gcc)
	$(CC) -o hello $(OBJECTS) $(LIBS_FOR_GCC)
else
	$(CC) -o hello $(OBJECTS) $(LIBS_FOR_OTHERCC)
endif
```

- **変数代入の条件分岐**

```
ifeq ($(CC),gcc)
 LIBS=$(LIBS_FOR_GCC)
else
 LIBS=$(LIBS_FOR_OTHERCC)
endif
```

- **利用可能なディレクティブ**
  - **ifeq, ifneq, ifdef, ifndef**

関数

- **変数と似た参照方法**

```
VALUE=$(subst xx,yy,aaxxbb)
```
VALUEにaayybbが代入される

```
CONTENTS=$(shell cat data.txt)
```
CONTENTSにはdata.txtの中身が代入される

```
SECOND=$(word 2, This is a pen)
```
SECOND=isと同じ

```
CDR=$(wordlist 2,$(words $(LIST)), $(LIST))
```
CDRには$LISTの2番目以降の単語のリストが代入される

- **他の関数の例**
  - **dir, notdir: シェルのdirname, basenameに似た動作**
  - **suffix, basename: 拡張子とそれ以外の部分に分ける**
    - **シェルのbasenameとは違う**
  - **wildcard: ワイルドカードを展開**

特殊な変数

- **ターゲット名や依存ファイル名などに展開される特殊な変数がある**

$@	ターゲット名
$<	最初の依存ファイル
$?	ターゲットより新しい依存ファイル
$+	すべての依存ファイル

```
hello: hello.o main.o
	gcc -o hello ¥
	   hello.o main.o
hello.o: hello.c
	gcc -c hello.c
main.o: main.c
	gcc -c main.c
```

```
CC=gcc
OBJECTS=hello.o main.o
hello: $(OBJECTS)
	$(CC) -o $@ $+
hello.o: hello.c
	$(CC) -c $<
main.o: main.c
	$(CC) -c $<
```

型ルール

- **指定したパターンにマッチしたらコマンドを実行する**

```
%.o : %.c
	$(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

  - *****.oは***.cに依存する**

```
hello: hello.o main.o
	gcc -o hello hello.o main.o
hello.o: hello.c
	gcc -c hello.c
main.o: main.c
	gcc -c main.c
```

```
CC=gcc
OBJECTS=hello.o main.o
hello: $(OBJECTS)
	$(CC) -o $@ $+
%.o: %.c
	$(CC) -c $<
```

変数を使いこなす

```
DATE1  = $(shell date)
DATE2 := $(shell date)
DATE3  = `date`
```

- **DATE1**
  - 参照されるたびにdateが実行される
  - 実行されるタイミングは最初(アクションが実行される前)
- **DATE2**
  - 1度だけdateが実行される
  - 実行されるタイミングは最初
- **DATE3**
  - 最初は`date`という文字列が展開されるだけ
  - Dateが実行されるのは各アクションが実行されるとき

課題5-1

- `DATE4  := `date``
  - はどれと同じ動作になるか
- **以下のようなルールで、2つの出力に違いがあるのはどれか、また、違いがないものは、どのような場合に違いが出るのか説明せよ**

```
test:
	echo $(DATE1)
	sleep 1
	echo $(DATE1)
```

課題5-2

- **wildcard関数を使用してみる**
  - 入力データの中から、2009年8月と9月のデータだけを処理したい
  - ファイル名に日付が含まれている
  - 出力データは拡張子を.inから.outに変えたものにする

makeの並列処理への応用

並列make

- `make -j` による並列化
  - 同時実行可能なコマンドを見つけて並列に実行
  - 依存関係の解析はmakeが自動的に行ってくれる

```
all: a b

a: a.c
	$(CC) a.c -o a
b: b.c
	$(CC) b.c -o b
```

同時実行可能

並列処理への応用

- makeは本質的にはワークフロー言語とその実行エンジン
  - コンパイル以外にもいろいろなことができる
- makeを使うとうれしいこと
  - 実行するコマンドの依存関係を簡単に記述可能
  - 簡単な並列化
    - 依存関係の解析はmakeが自動的に行ってくれる
  - 耐故障性
    - 途中で失敗しても、make し直せば続きからやってくれる

課題6 (1ノードの例)

- **Makefile**

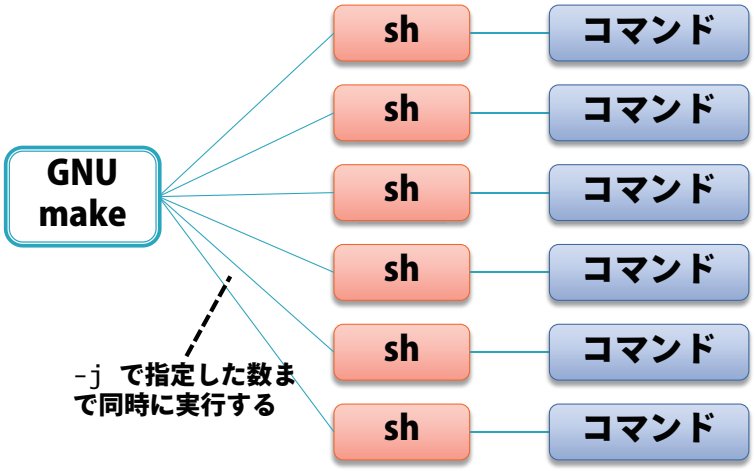
```
FILE_IDS := $(shell seq 1 10)
FILES    := $(FILE_IDS:%=%.dat)

all: $(FILES)

%.dat:
	sleep 5
	touch $@
```

  - 変数や%を使わない場合どのようなMakefileになるか
  - makeとmake -jの実行時間を比較せよ

make -j

- `make -j` **最大並列度**
  - 最大並列度で指定した数まで同時にコマンドを実行する
  - 最大並列度の最大値は4096
    - RHEL5における制約
    - それ以上を指定すると1を指定したものとみなされる
  - 最大並列度を指定しない場合は可能な限り同時にコマンドを実行する

並列makeの動作の仕組み



複数ノードで並列make

- HA8000クラスタシステムの場合、1ノードでは使えるCPUコア数は16まで
- 多数のノードを使用すれば、よりたくさんの処理を行うことが可能
- GXP makeを使用すると複数ノードで並列makeを実行可能
  - GXP makeは並列シェルGXPと一緒に配布されているソフトウェア
  - makeの処理を、マスターワーカー型の並列処理として複数ノードで実行可能
  - 各ノードでファイルが共有されていることが前提

GXP

- 並列分散環境を簡単に扱うための、並列版シェル
  - 多数のノードのインタラクティブな利用
  - 並列ワークフローの実行(GXP make)
- 詳しい情報

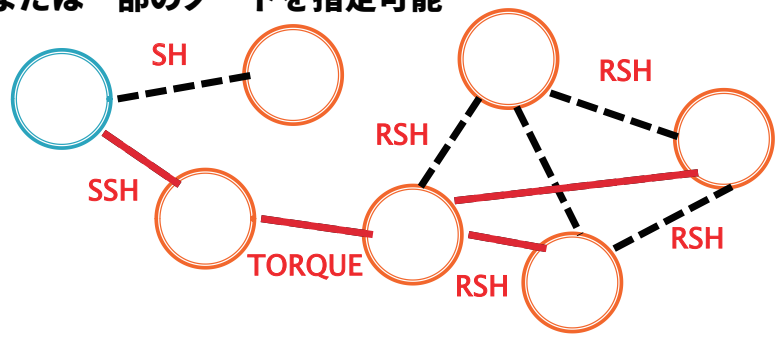
  http://www.logos.t.u-tokyo.ac.jp/gxp
  http://sourceforge.net/projects/gxp
- ダウンロード方法

```
$ cd /nfs/all/$USER
$ cvs -d ¥
   :pserver:anonymous@gxp.cvs.sourceforge.net:/cvsroot/ ¥
   gxp co gxp3
```

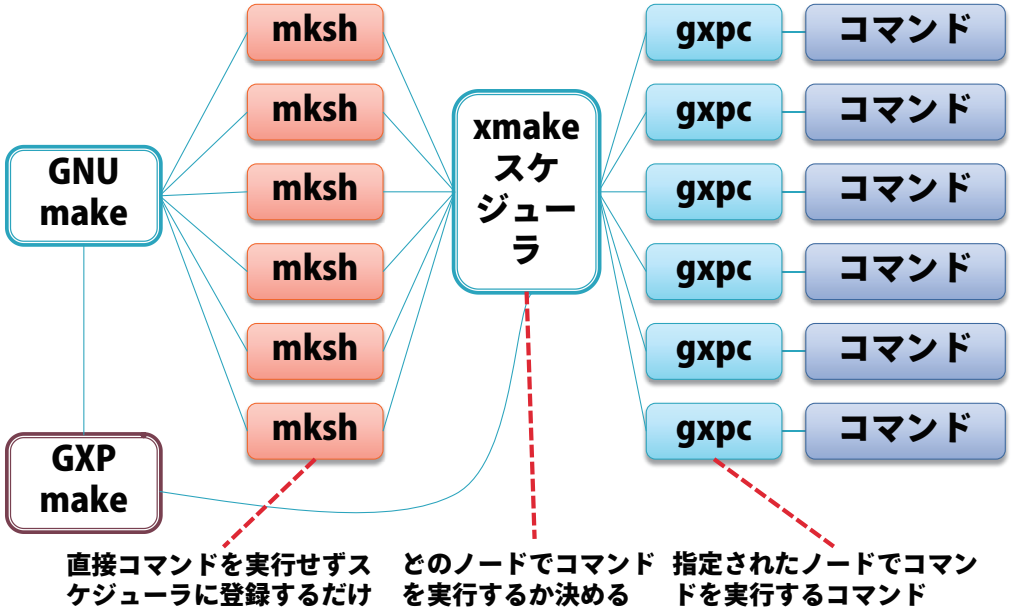
GXPの動作

- 各計算ノードでデーモンプロセス(GXPD)を起動
  - ノード集合と、GXPDの起動方法を指定(use)
    - SSH, PBS, GridEngine等が利用可能。拡張も可能
  - ノード集合を指定して、GXPDを起動(explore)
- e(execute)コマンドでユーザプロセスを起動
  - 全部または一部のノードを指定可能

use
explore
execute



GXP makeの動作の仕組み



サンプルスクリプト

```
#@$-N 4
#@$-J T1

NODES=`qstat -f $PBS_JOBID|grep Node:|awk '{print$2}'`
MAX_NODEID=`expr $NODES - 1`

gxpc --root_target_name head
gxpc use torque_psched head node
gxpc explore node[[1-$MAX_NODEID]]

echo cpu . 16 > gxpc_make.conf

cd $PBS_O_WORKDIR
gxpc cd `pwd`
gxpc make -j
gxpc quit
```

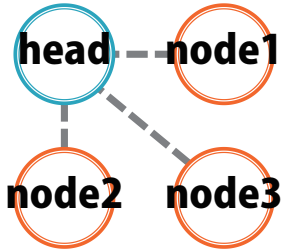
MAX_NODEIDに3を設定

GXPDの起動

torque_pschedはtorqueジョブ内のプロセス起動

ノードあたり16プロセス起動

並列makeの実行



GXP pp

- パラメタ並列に特化した機能
  - GXP makeのラッパとして実現
- 書式

```
gxpc pp [makeオプション] cmd='コマンド' parameters='a b' ¥
   a='値 値 …' b='値 値 …'
```

  - コマンドの中で、a, b, … を、$(a), $(b), … で参照
  - パラメタにa, b, c,…を使う場合は、parameters=を省略可能
- 例 (10 * 3=30個のコマンドを並列実行)

```
gxpc pp -j cmd='echo $(a) $(b)' a="`seq -s ¥  0 9`"  ¥
   b="`seq  -s ¥  0 2`"
```

MapReduce

- MapReduceモデル
  - Googleが提案する、大規模データの並列処理に特化したプログラミングモデル
  - 1レコードに対する処理を書くと、処理系が大規模データに並列適用
  - 入力データIは、レコードの集合
  - プログラムは、以下の2つの処理を定義
    - Map: レコード→(key, value)の集合
    - Reduce: (key, value)の集合→出力
    - 異なるレコードに対するmap処理と、異なるkeyに対するreduce処理が並列実行可能

GXP mapred

- GXP make上に構築されたMapReduce処理系
- GXPが動く環境ならどこでも動く
- カスタマイズが容易
  - Makefileといくつかの小さなスクリプト
- 例(word count)
  - Mapper: レコード→(単語1, 1),(単語2, 1),…
  - Reducer: それぞれのkeyについてvalueの和を出力

```
gxpc mapred -j input=入力ファイル output=出力ファイル ¥
  mapper=Mapper reducer=Reducer
```

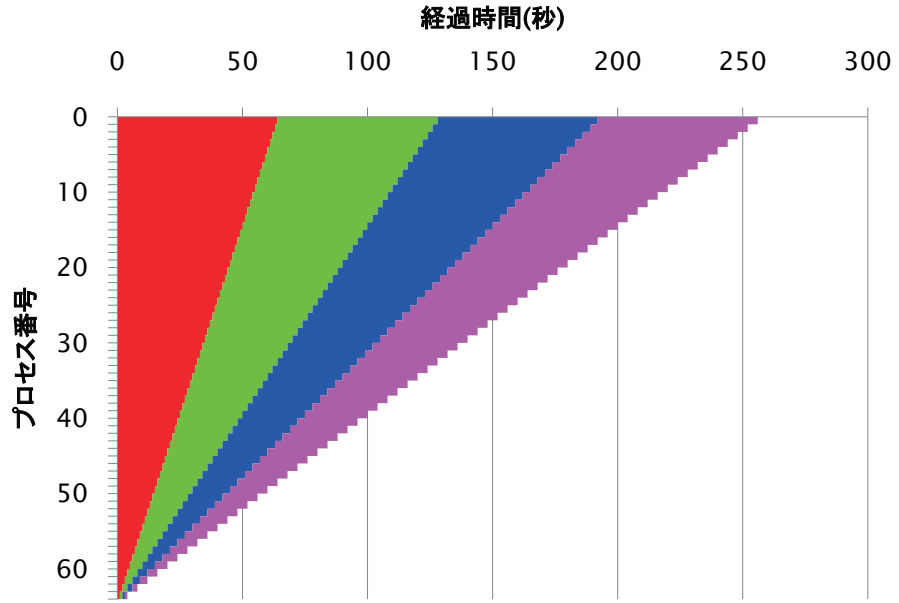
GXP mapred

- GXP mapred よく使うオプション
  - -n : make と同じdryrun. 何が起こるのかをチェック
  - -j : make と同じ並列実行
  - int_dir=中間ファイル用ディレクトリ名
  - n_mappers=map ワーカ数
  - n_reducers=reduce ワーカ数
  - keep_intermediates=y : 中間ファイルを消さない
  - small_step=y : 細かいステップでの実行
  - dbg=y : 上記二つの略記
    - 注: (n_mappers*n_reducers) 個の中間ファイルが作られる

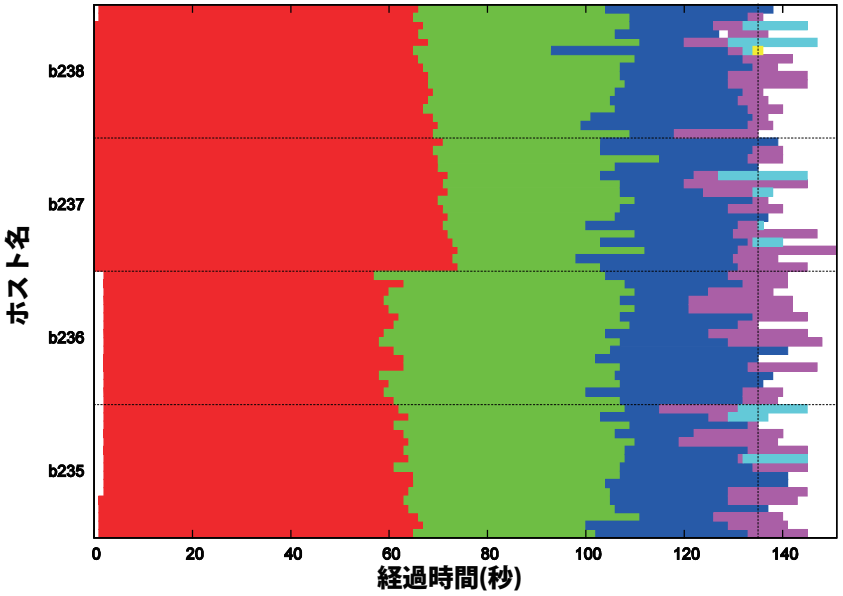
課題7

- 以下に述べる並列処理を実行せよ
- 処理の内容
  - 複数の入力ファイルがある
  - 入力ファイルごとに、その内容に従って処理を行い、1つの出力ファイルを生成する
    - 入力ファイルの内容により、処理時間は異なる
  - それぞれのタスクは独立で、並列実行可能
- 以下のそれぞれの場合を実際に試して、実行時間の違いの理由を考えよ
  - 処理するファイルをプロセスごとに固定する場合(MPI)
  - マスターワーカー型の負荷分散を行う場合(GXP make)

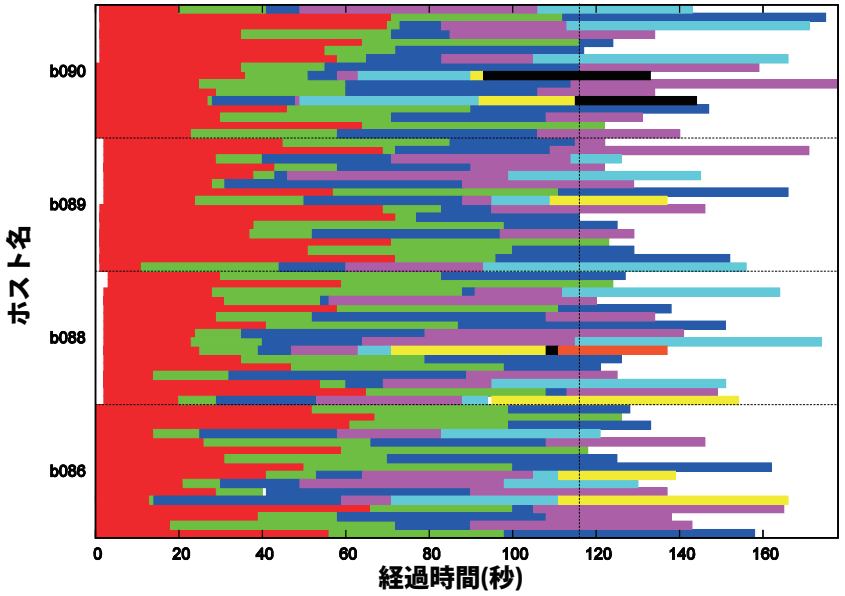
負荷分散を行わない場合



負荷分散を行った場合



負荷分散あり・ランダムな順番



まとめ

- **ファイルシステムやバッチキューイングシステム**
  - **HA8000クラスタシステムに固有の情報を活用することで、より効率的なシステムの利用が可能**
- **Makefileの基礎**
  - **Makefile、makeを利用することで、変更箇所だけを再作成する分割コンパイルが可能**
- **並列ワークフロー処理の基礎**
  - **makeを拡張したGXP makeを利用することで、大規模な並列処理を実行可能**